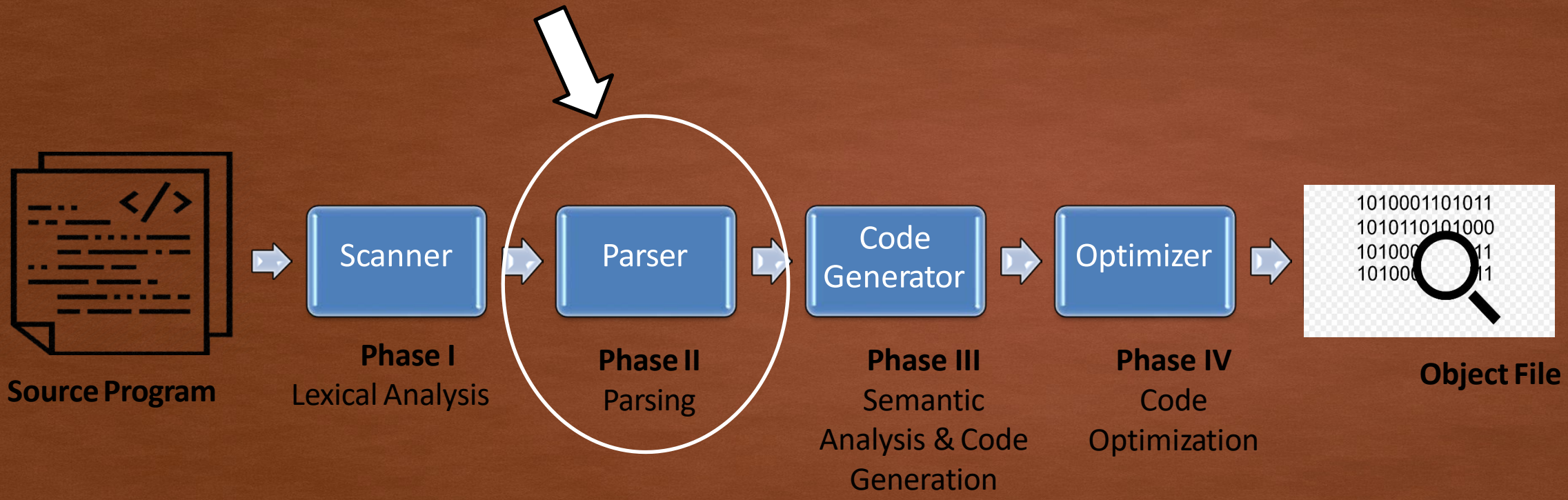
Phase II : Parsing: First Principles



PARSING- RECAP

The parse tree has as its leaves the terminals (tokens returned by the lexer).

The parser uses the rules of the grammar to combine terminals/non-terminals into other non-terminals, or transform non-terminals into other non-terminals.

Each such step of combining/transforming terminals/non-terminals into other non-terminals is called a ***production.***

Eventually, there should only one non-terminal left: the *root* representing the ***goal symbol***

If the parser cannot build such a tree, it concludes that the code has a ***syntax error***.

Parsing Example

```
a1 = x + y
```

Aim: Check whether this list of tokens gives a *syntactically correct statement* (aka the *goal symbol).*

We will assume here that we are using a BNF grammar.

Our *goal symbol* in the example here is a single assignment statement.

Building a Parse Tree : An Example

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Grammar of our Language in BNF

a1 = x + y

Input Program : A simple assignment statement

<symbol> = <symbol> + <symbol>

Tokens returned by the lexer

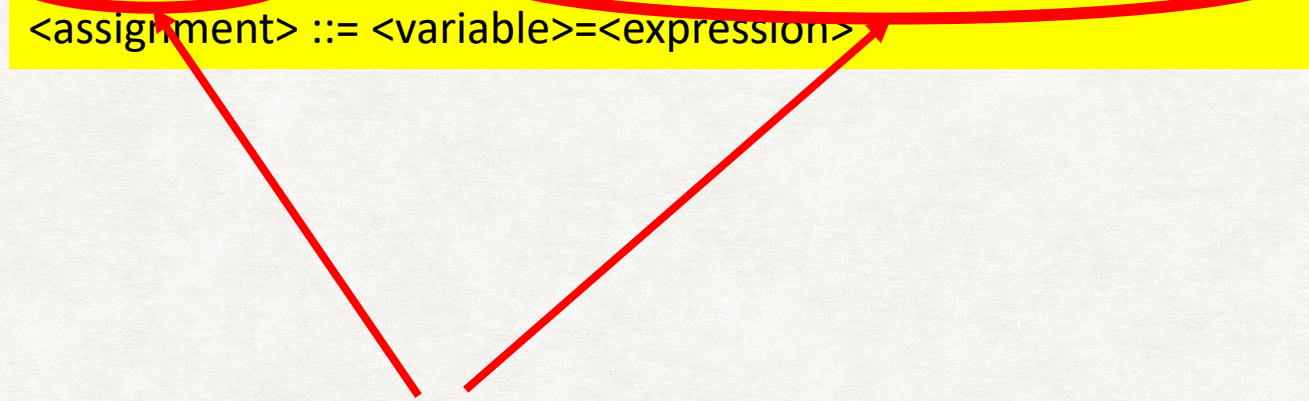
Building a Parse Tree

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

This rule enables productions that recognise a symbol as a variable

Building a Parse Tree

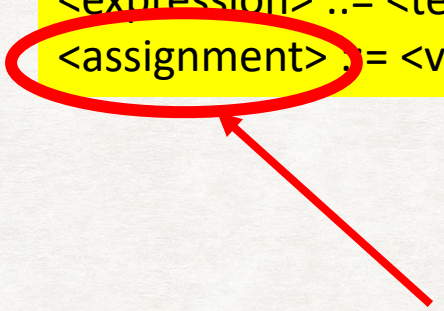
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>



This rule is an example of a **recursive definition**, where the category <expression> is defined in terms of its own self.
Hence the symbol <expression> appears on both the LHS and RHS of the rule.
It enables productions that combine two expressions and an operator between them into an expression.

Building a Parse Tree

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>



Goal symbol (we want this to be at the root of the tree we will build)

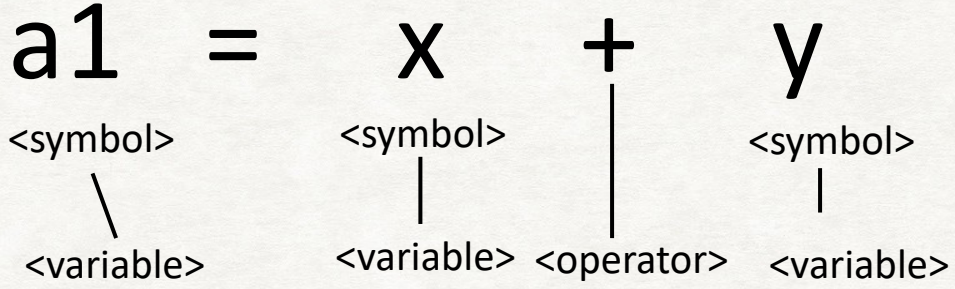
Building a Parse Tree : An Example

```
a1  =   x   +   y
```

<symbol> <symbol> <symbol>

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example



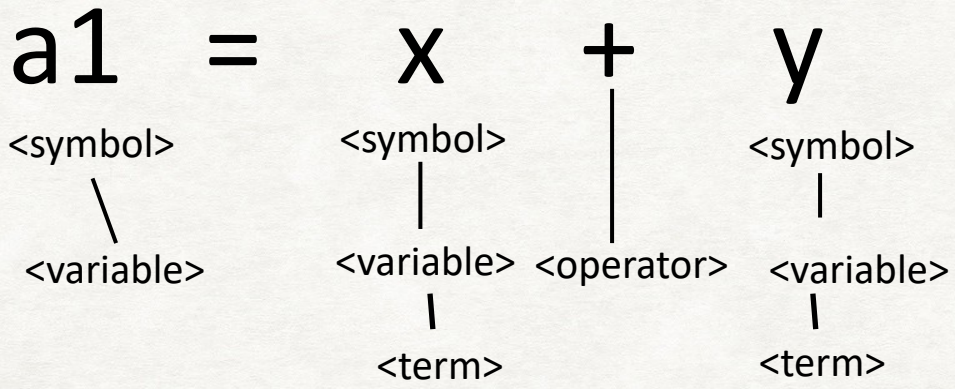
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example

a1	=	x	+	y
<variable>		<variable>	<operator>	<variable>

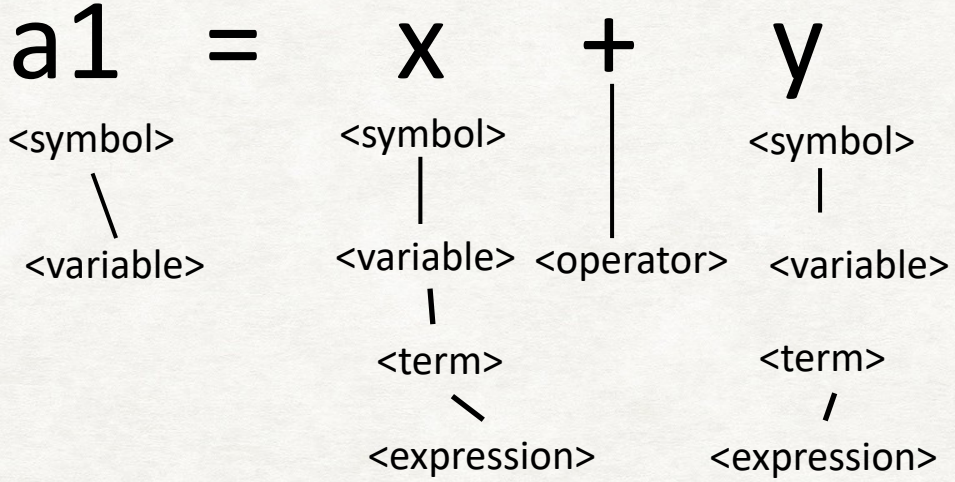
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example



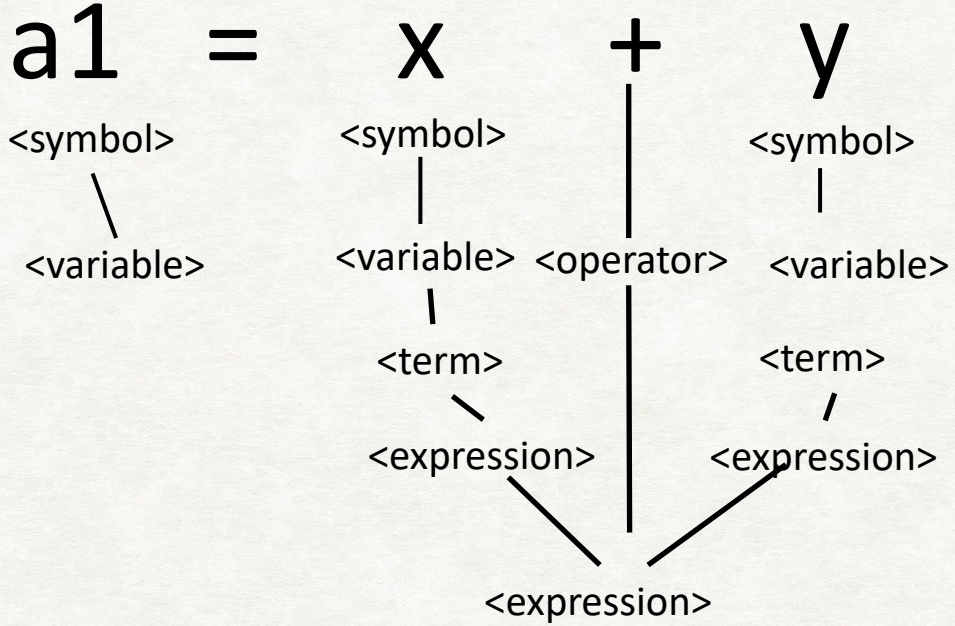
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example



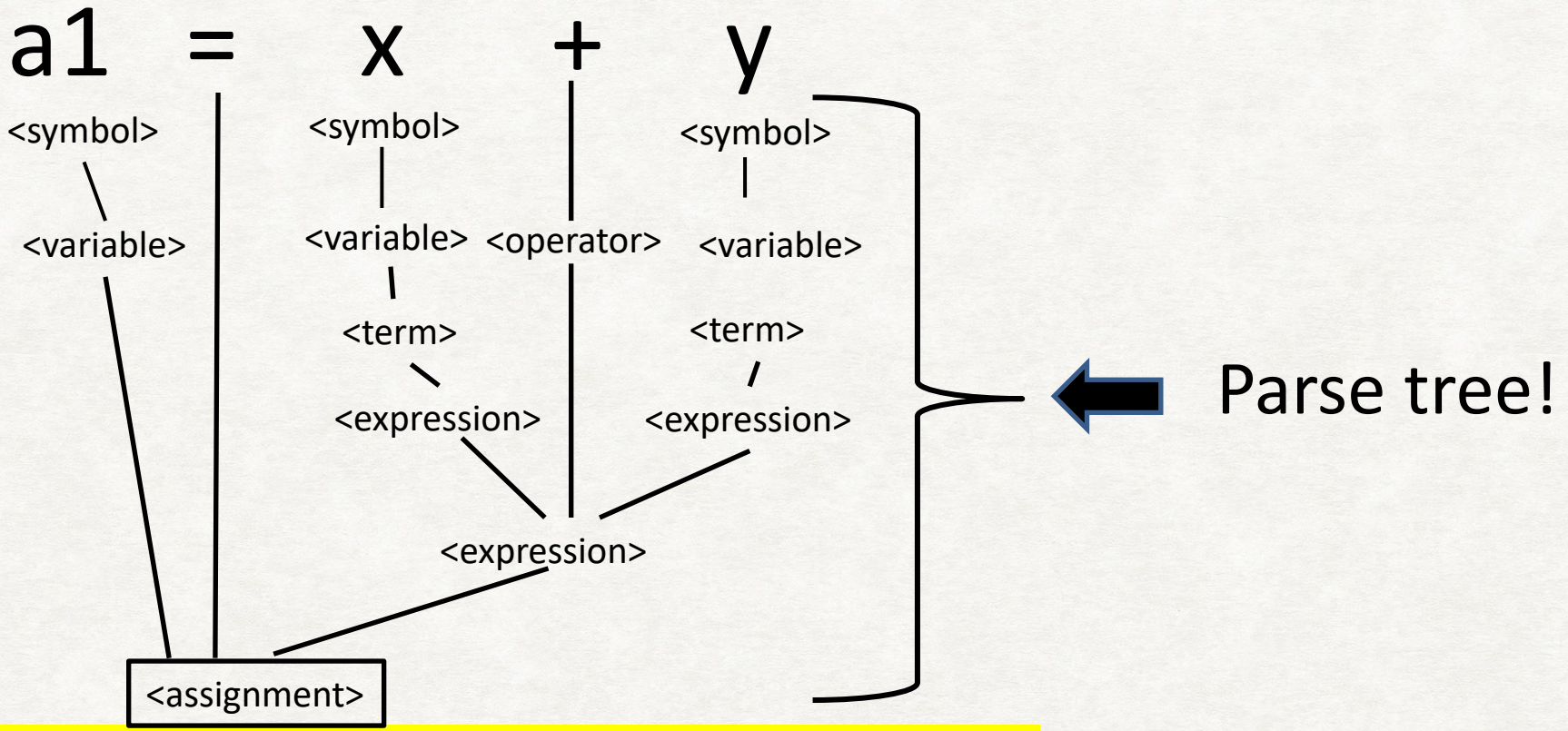
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Building a Parse Tree : An Example



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

PRACTICE EXERCISE

Given the grammar for an if-else statement, check whether the following if statement is syntactically correct or not.

```
if ( x == y) x = z; else x = y;
```

Number	Rule
1	<if statement> ::= if (<Boolean expression>) <assignment statement> ; <else clause>
2	<Boolean expression> ::= <variable> | <variable> <relational> <variable>
3	<relational> ::= == | < | >
4	<variable> ::= *x* | *y* | *z*
5	<else clause> ::= else <assignment statement> ; | Λ
6	<assignment statement> ::= <variable> = <expression>
7	<expression> ::= <variable> | <expression> + <variable>

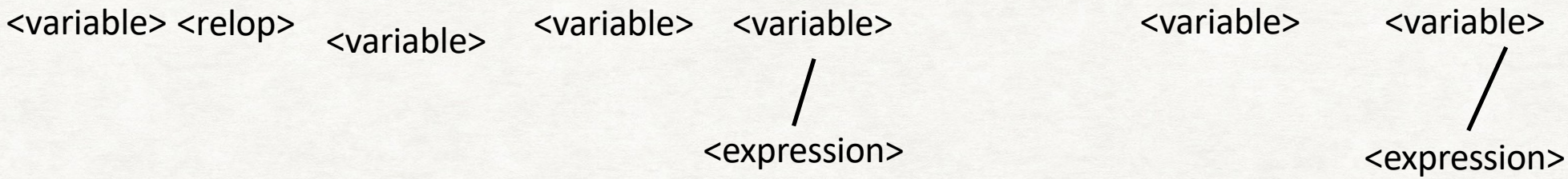
Grammar for a simplified version of an *if-else* statement

if (x == y) x = z ; else x = y ;

<variable> <relop> <variable> <variable> <variable> <variable> <variable>

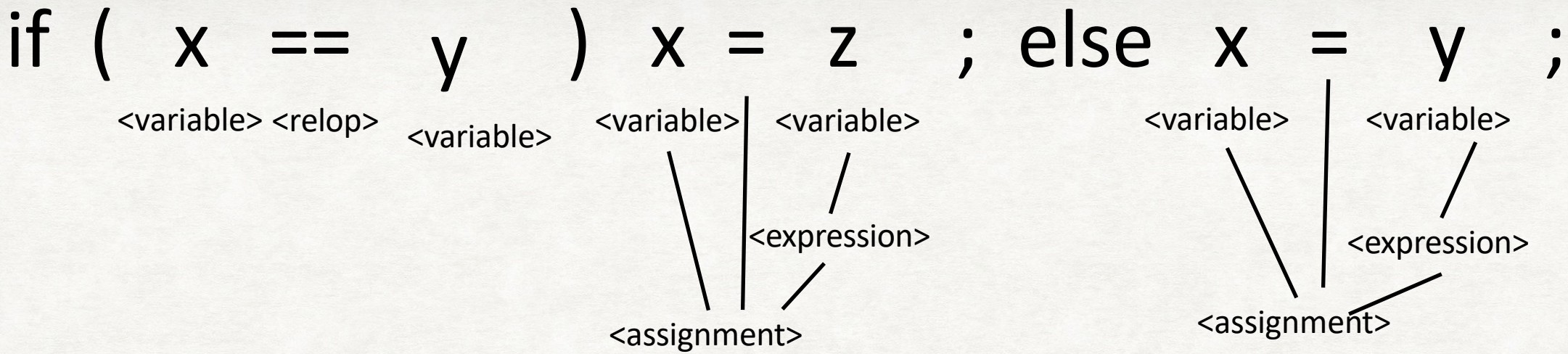
<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>

if (x == y) x = z ; else x = y ;

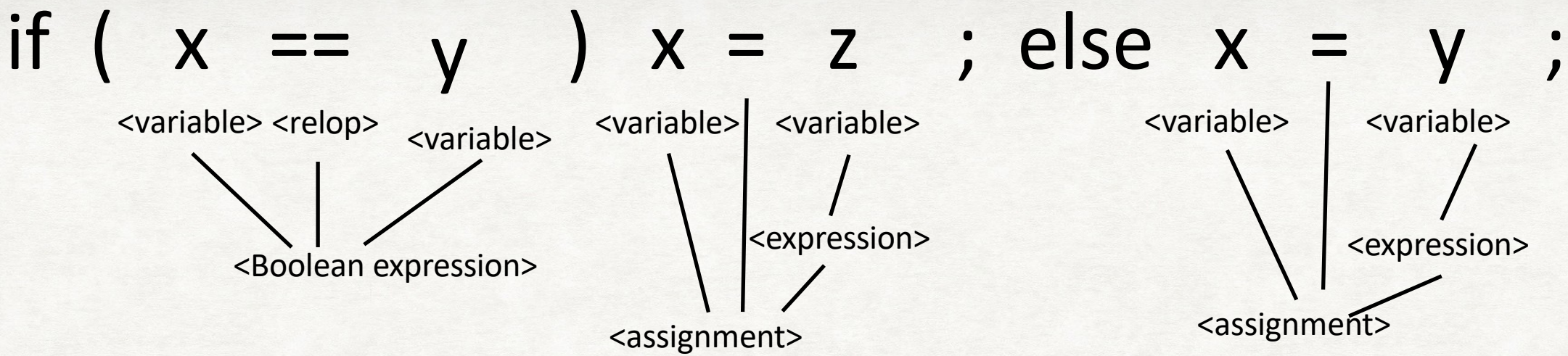


If the parser can **look ahead** it can often make the right choice.

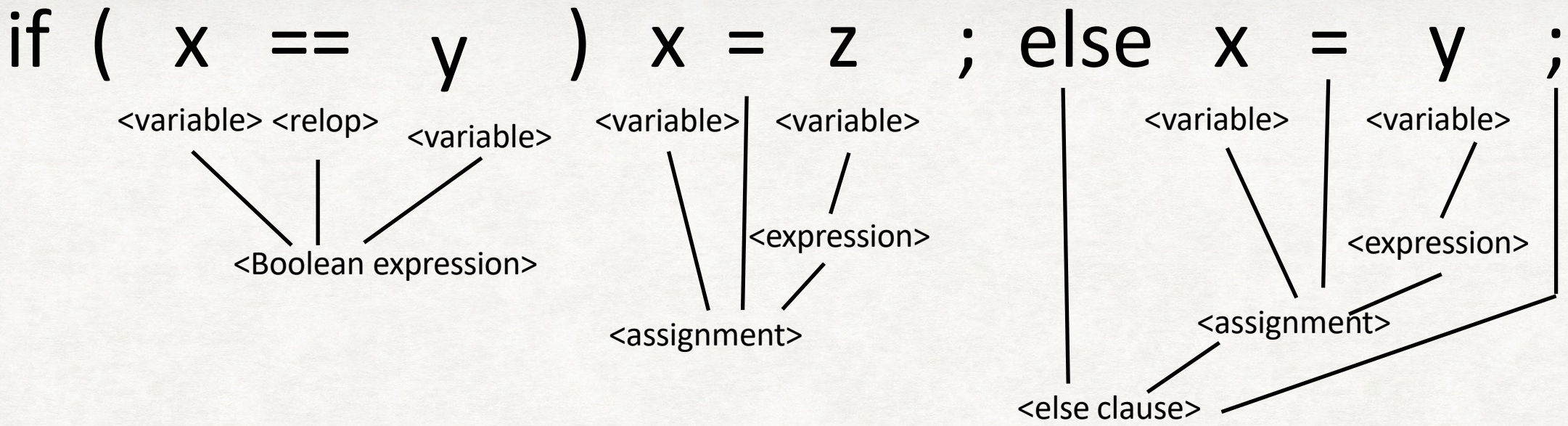
<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>



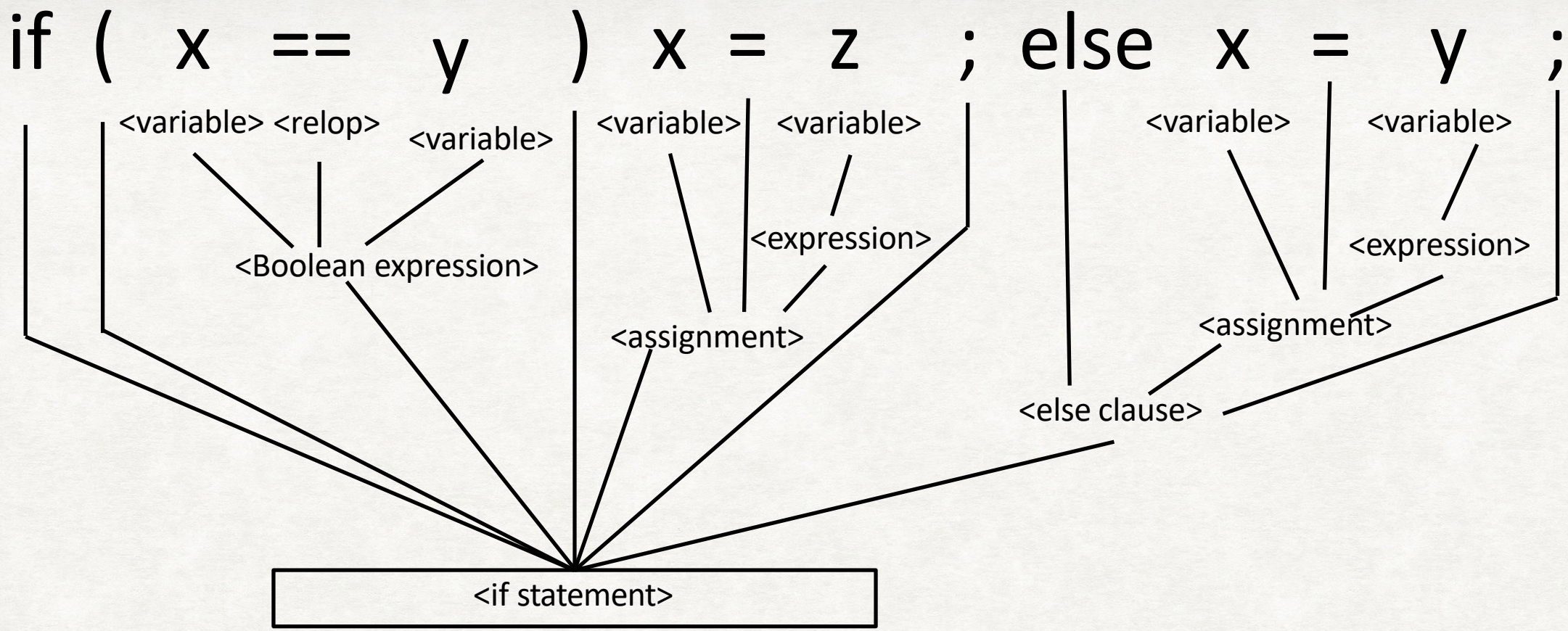
<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>



<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>

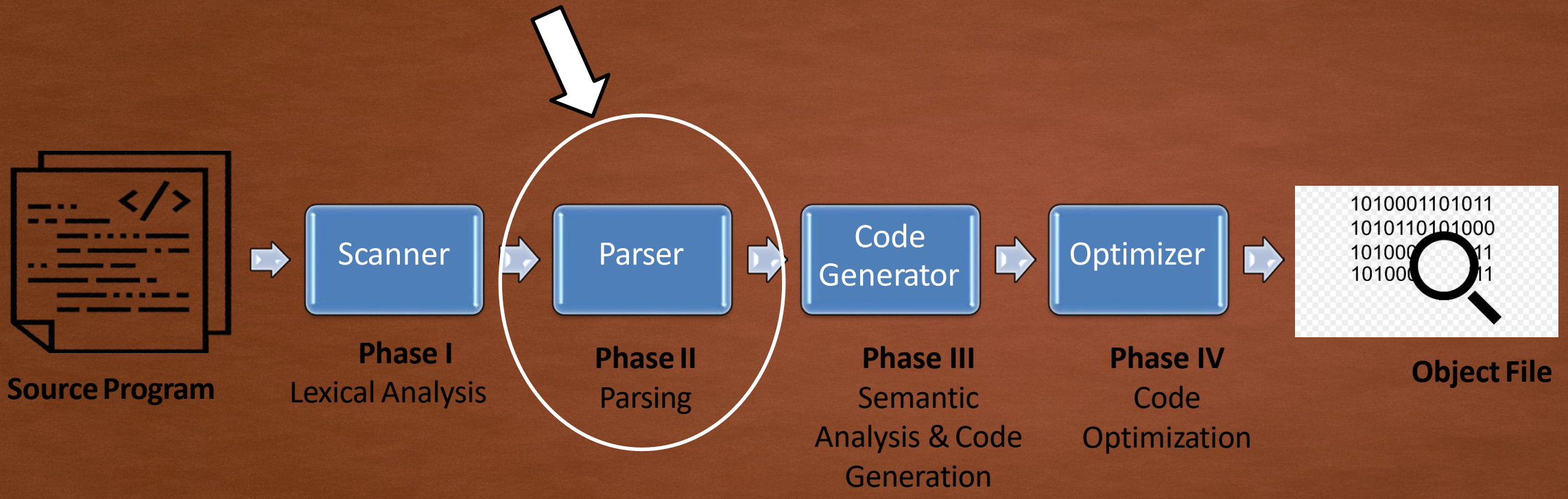


<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>

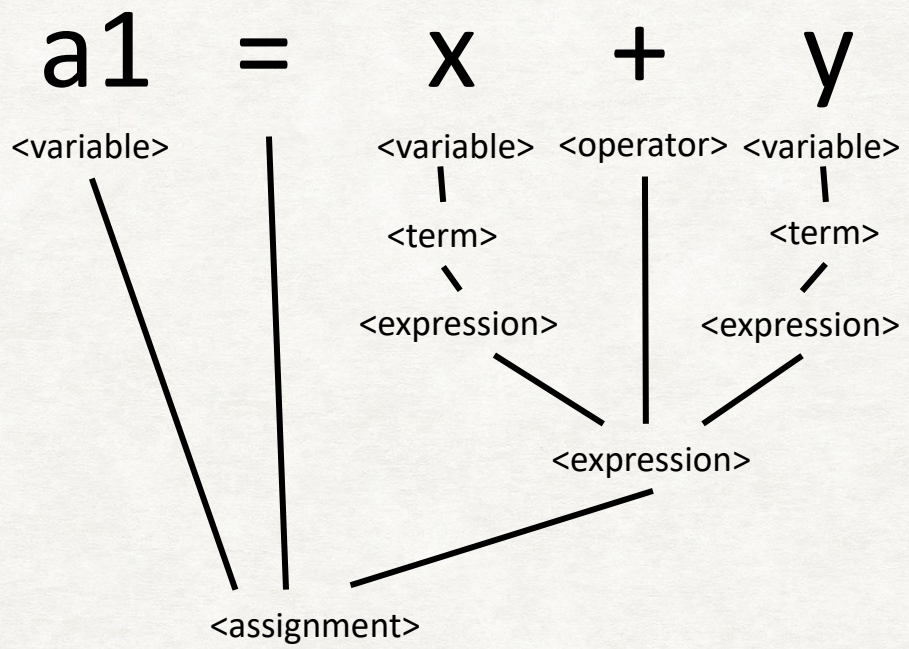


<if statement> ::= if(< Boolean expression>)< assignment>;<else clause>
<Boolean expression> ::= <variable>|<variable> <relop><variable>
< relop> ::= ==|<|>|
<variable> ::= x|y|z
< else clause> ::= else< assignment>;| Λ
<assignment> ::= <variable>=<expression>
<expression> ::= < variable>|<expression>+< variable>

Phase II : Parsing
Priority and Ambiguous Grammars

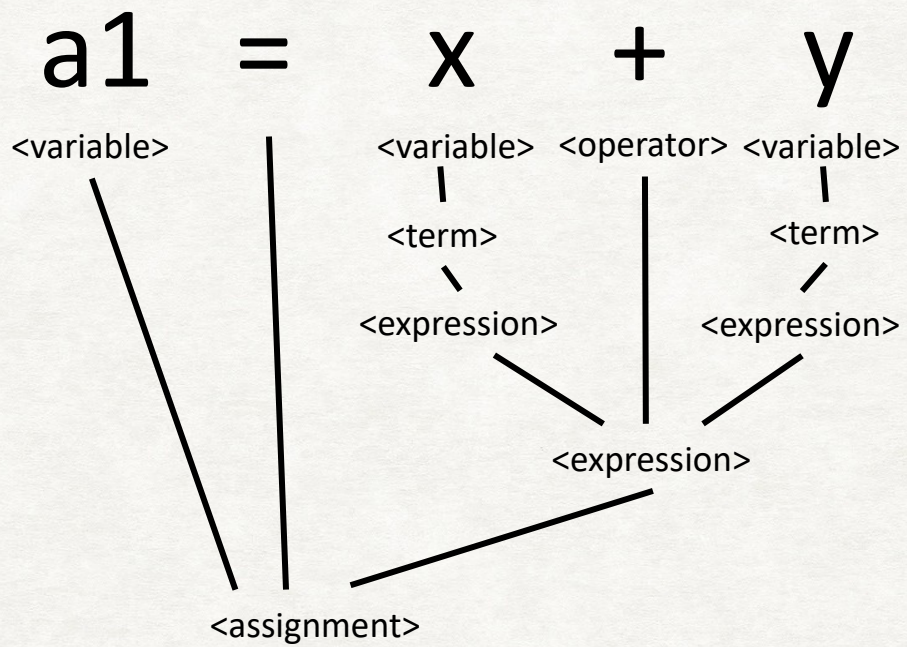


Recall the previous example: Simple assignment statement



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

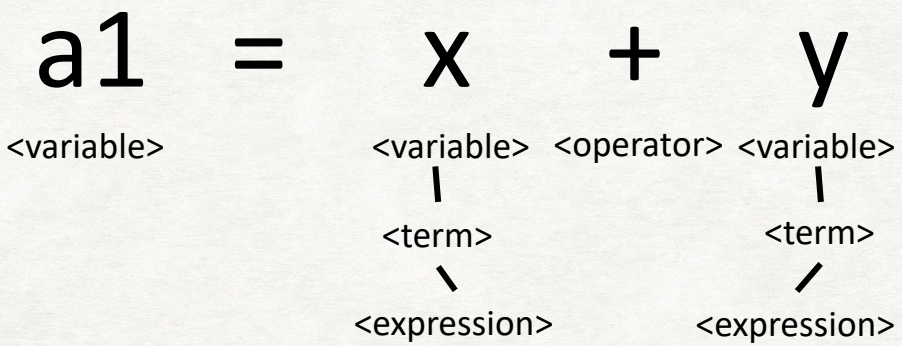
Example: Simple assignment statement



Could we have done this differently?

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Try again: Simple assignment statement

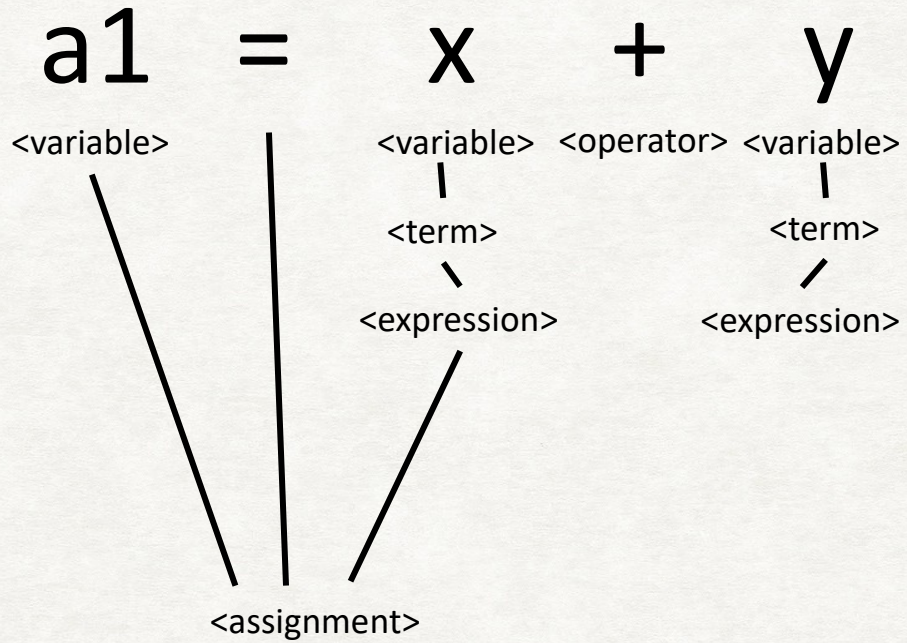


Could we have done this differently?

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

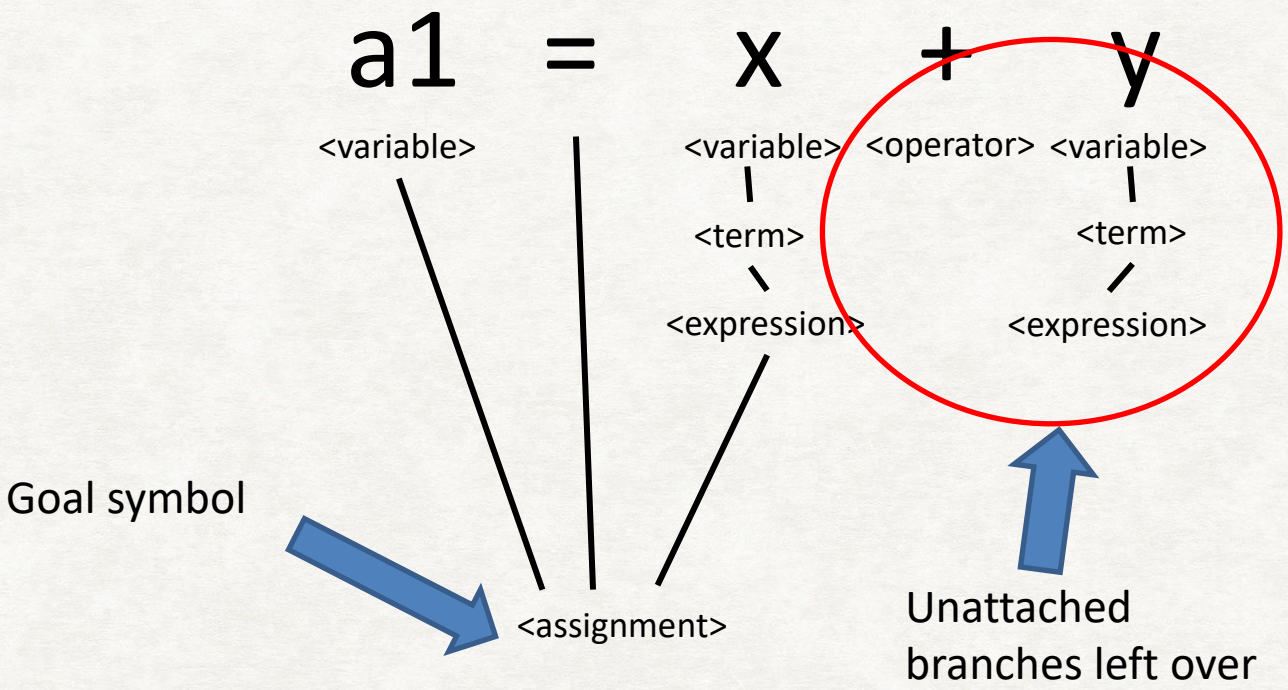
Try again: Simple assignment statement

Yes we could have!



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

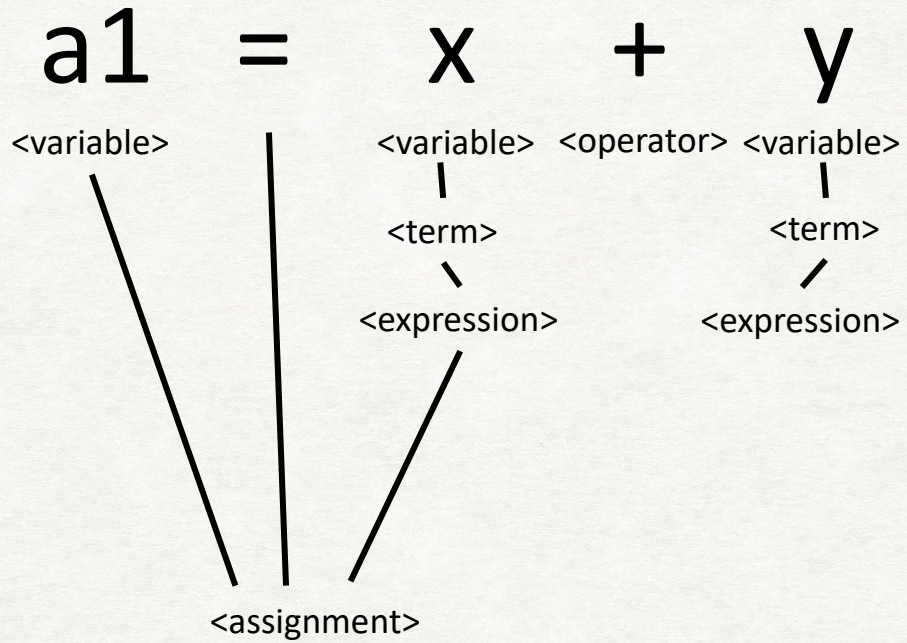
Try again: Simple assignment statement



But it doesn't work out!

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Try again: Simple assignment statement



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

The parser can detect this condition.

What does it do?

The parser will **backtrack** (undo the production) and try to match another grammar rule.

PARSING EXAMPLE 2

A more complex assignment statement

```
a1  =   x   +   y   *   z
```

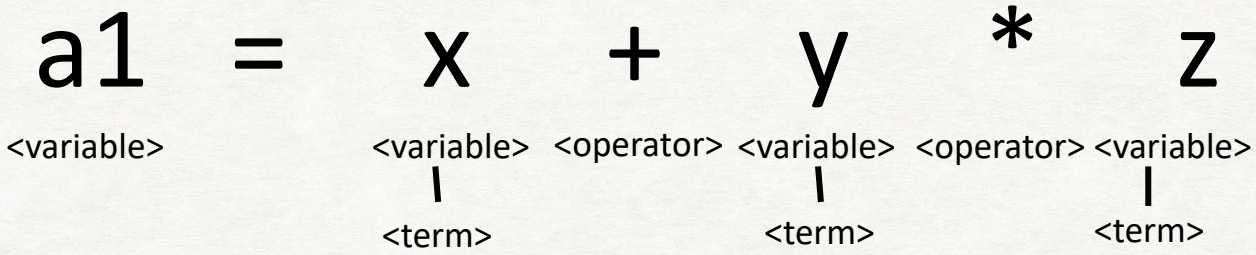
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

a1	=	x	+	y	*	z
<variable>		<variable>	<operator>	<variable>	<operator>	<variable>

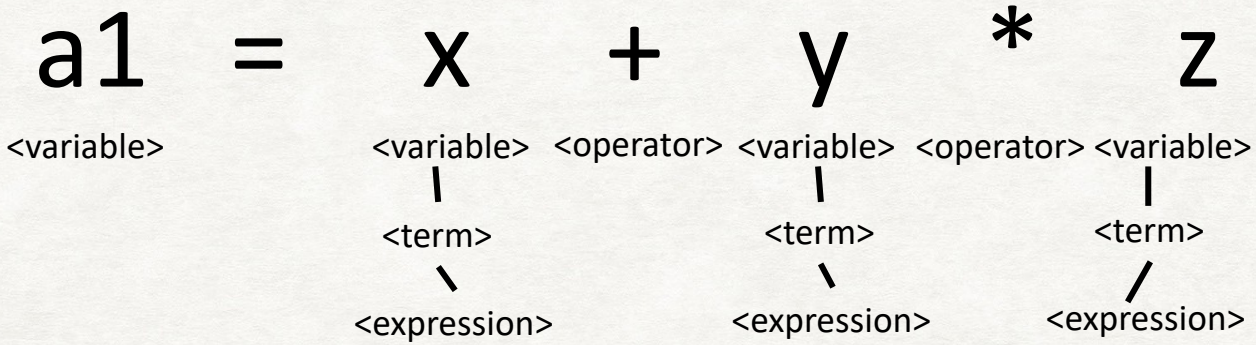
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement



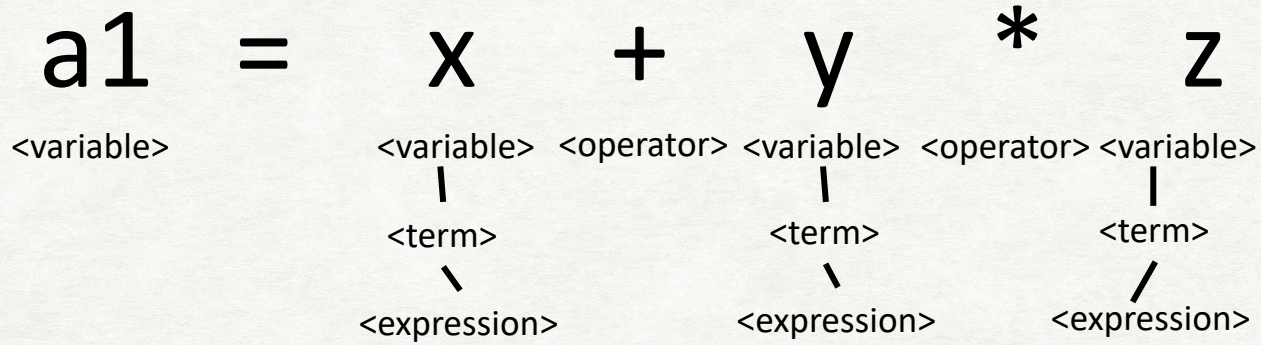
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

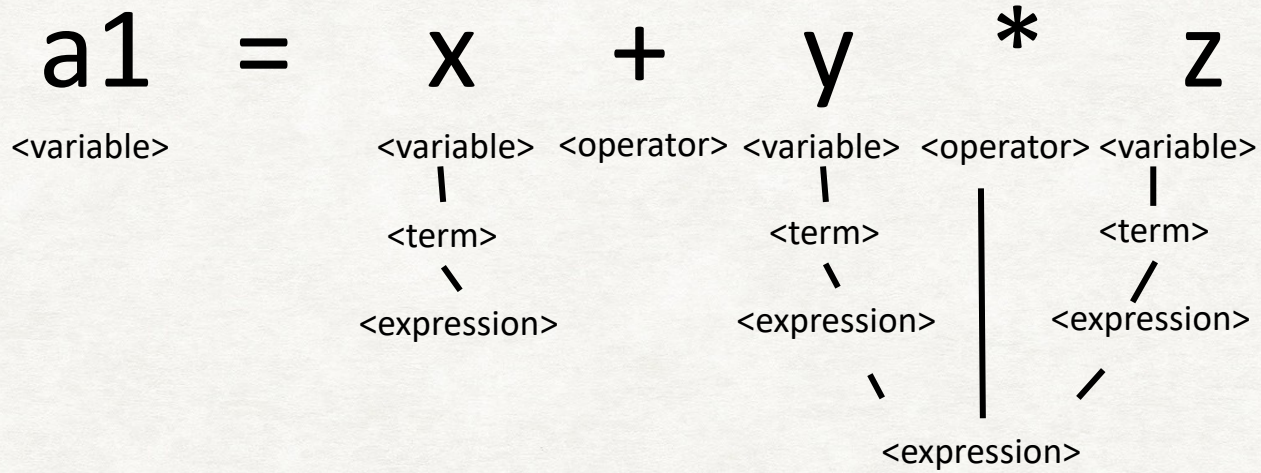
A more complex assignment statement



We now have two ways in which we can proceed:

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

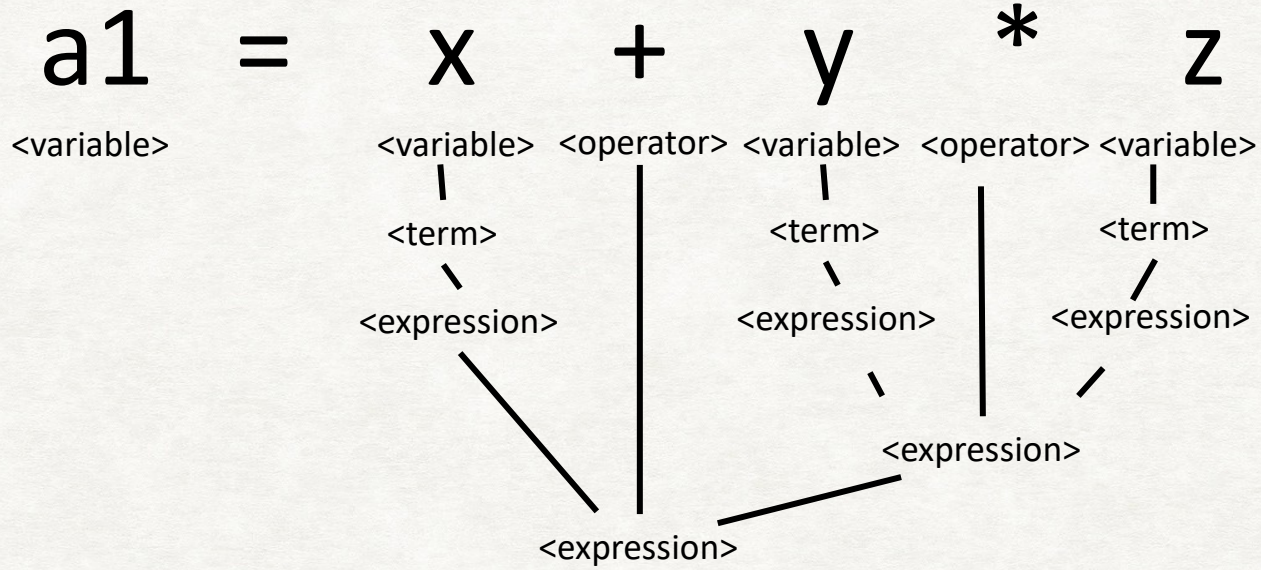


We now have two ways in which we can proceed:

1. Right to Left

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

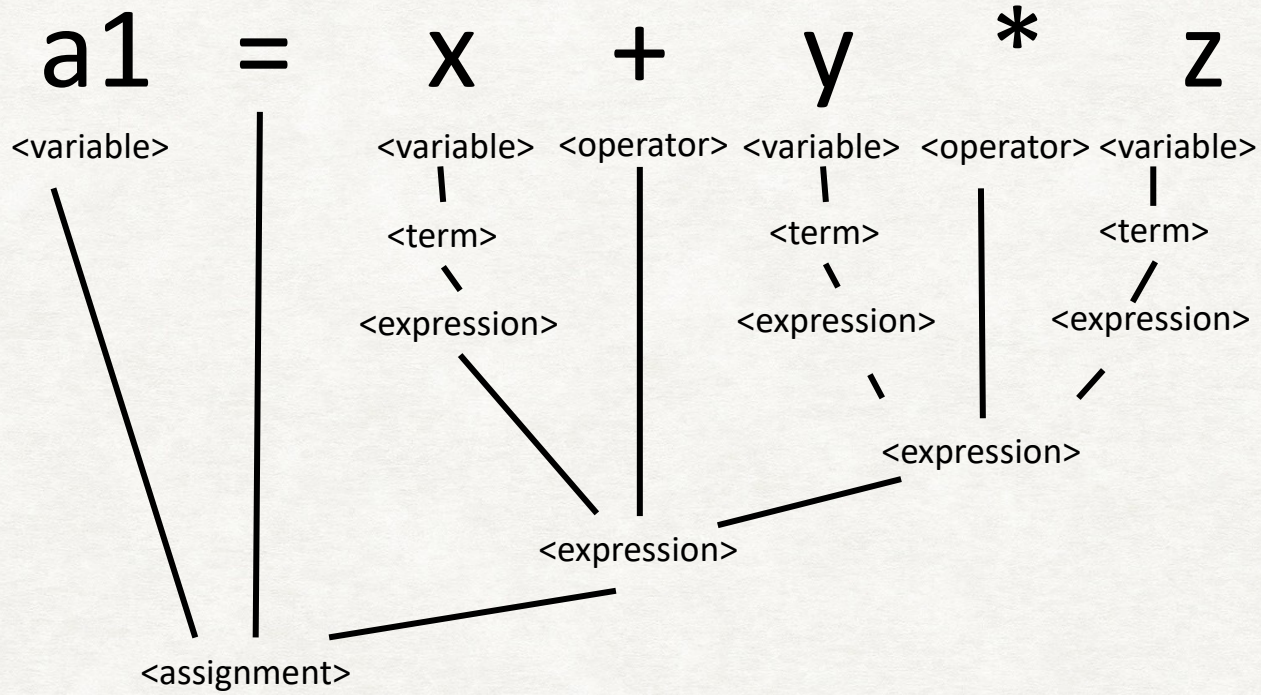


We now have two ways in which we can proceed:

1. Right to Left

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

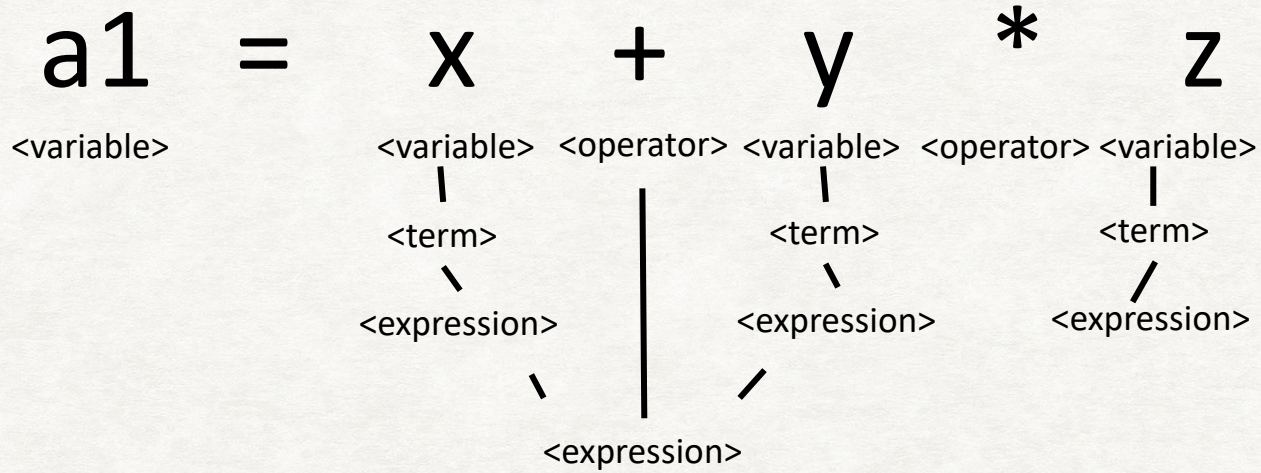


We now have two ways in which we can proceed:

1. Right to Left

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

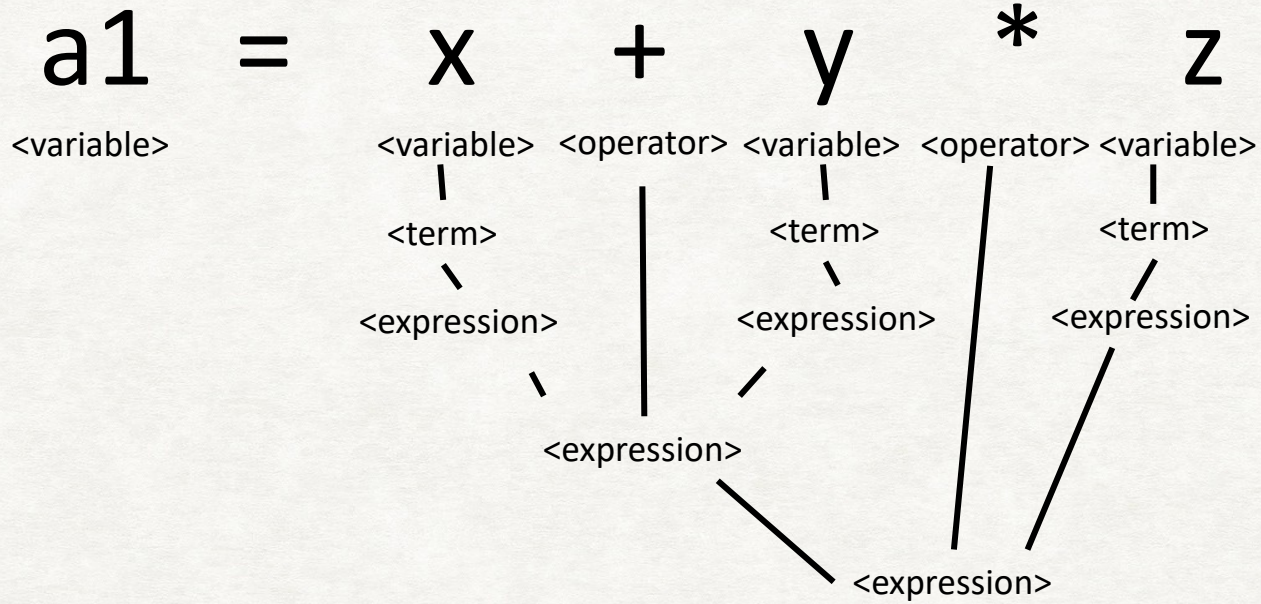


We now have two ways in which we can proceed:

2. Left to Right

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

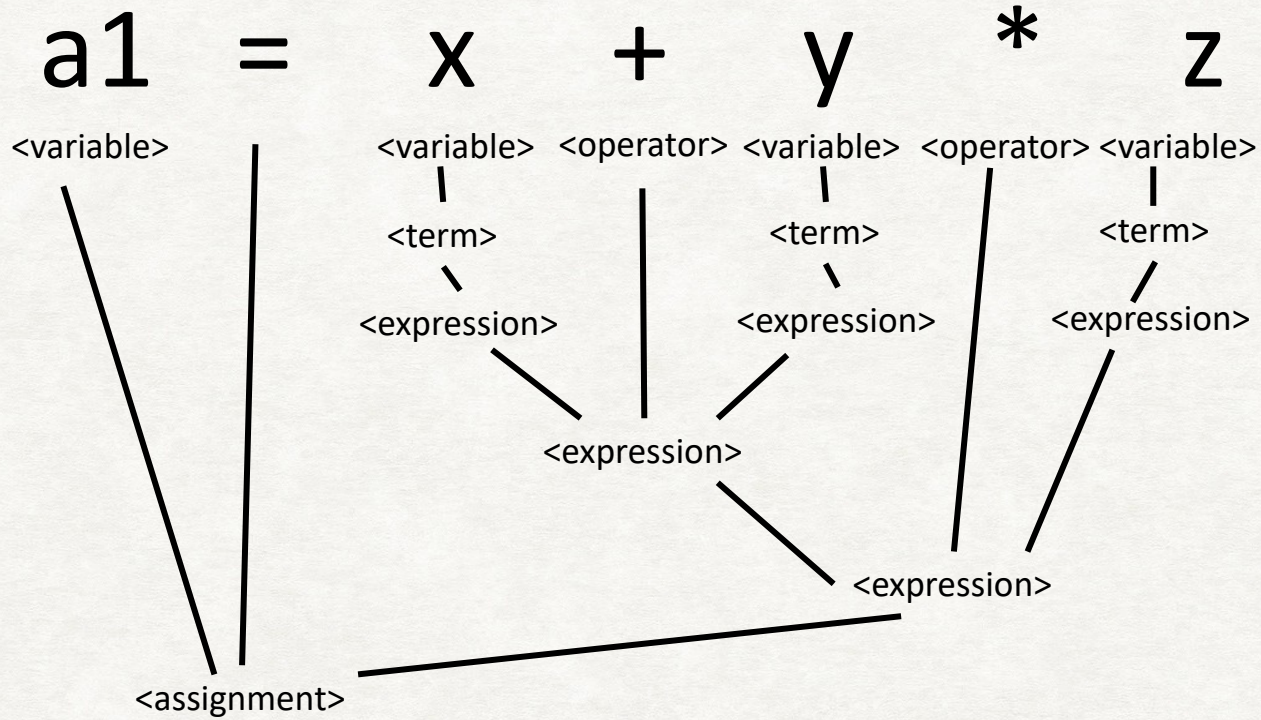


We now have two ways in which we can proceed:

2. Left to Right

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

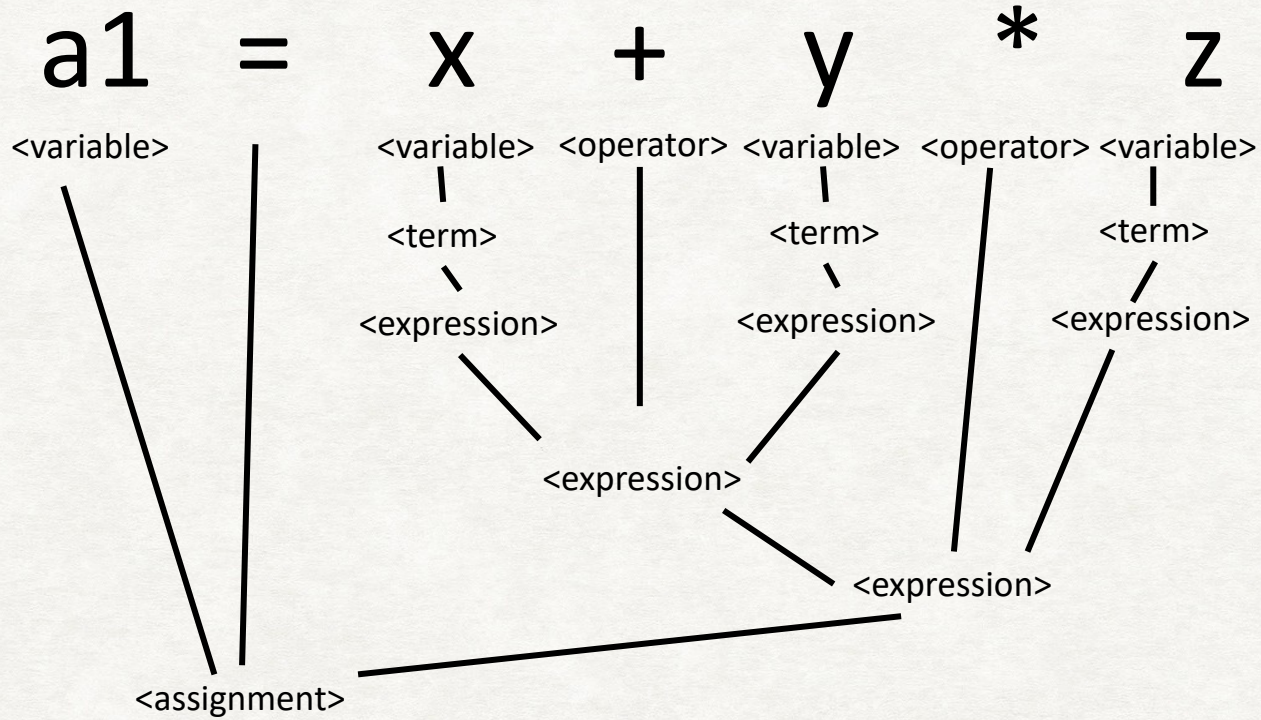


We now have two ways in which we can proceed:

2. Left to Right

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement

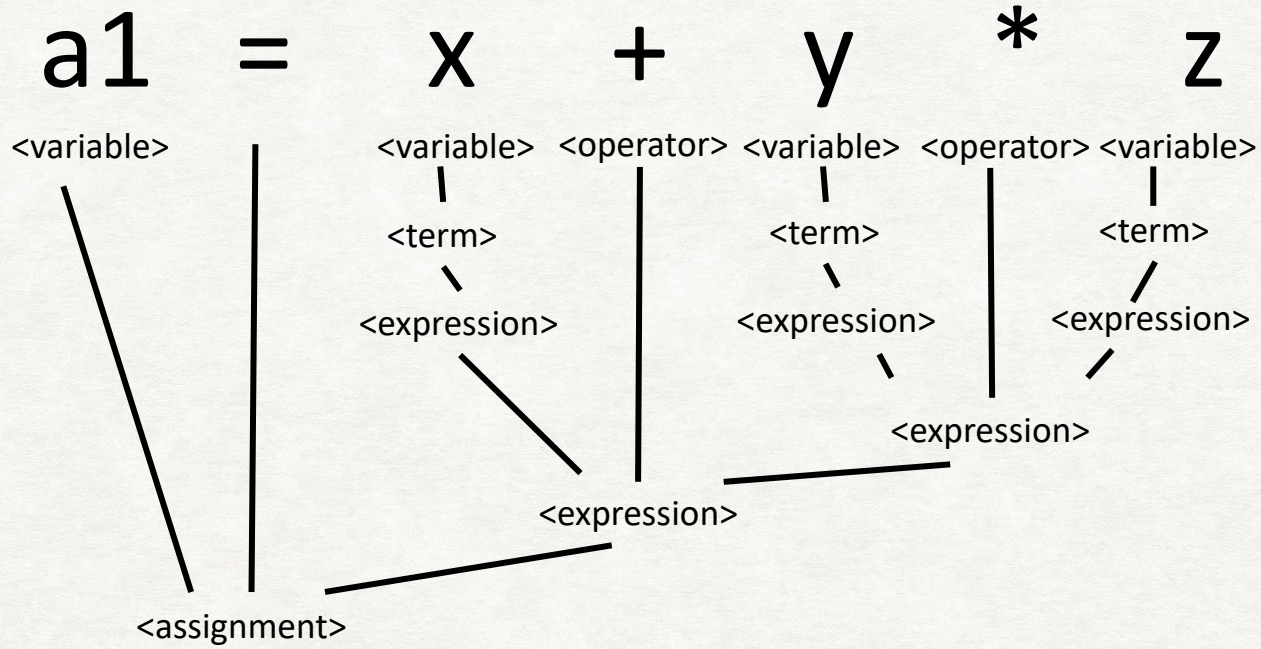


But Option 2 gives the wrong result!

It's equivalent to (x+y)*z.

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

A more complex assignment statement



Option 1 does what we want.

But our grammar gives no preference to either option 1 or 2.

It is *ambiguous*.

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

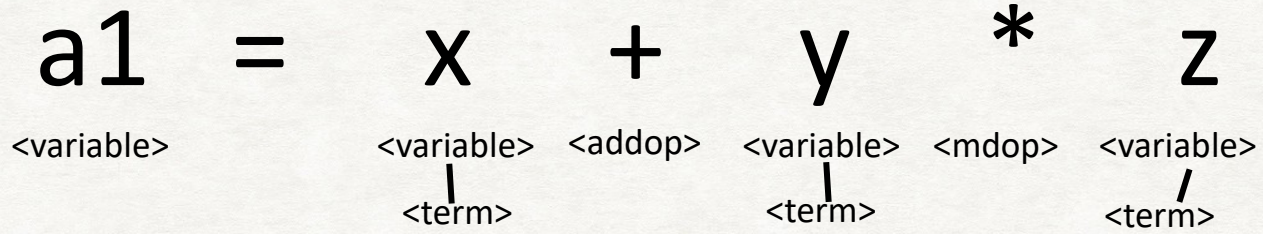
A grammar for multiplication before addition

a1 = x + y * z

The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

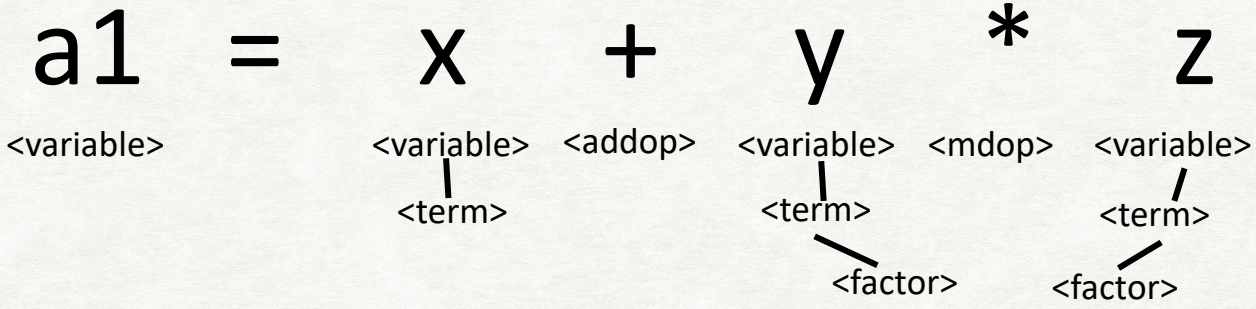
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

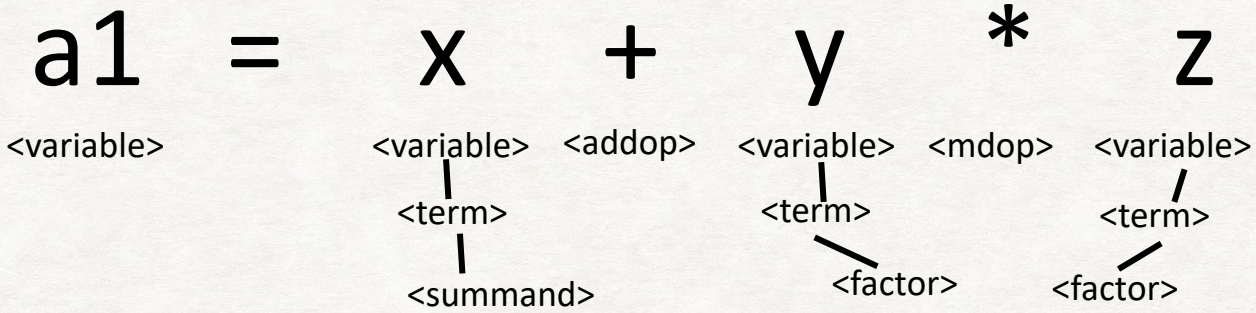
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

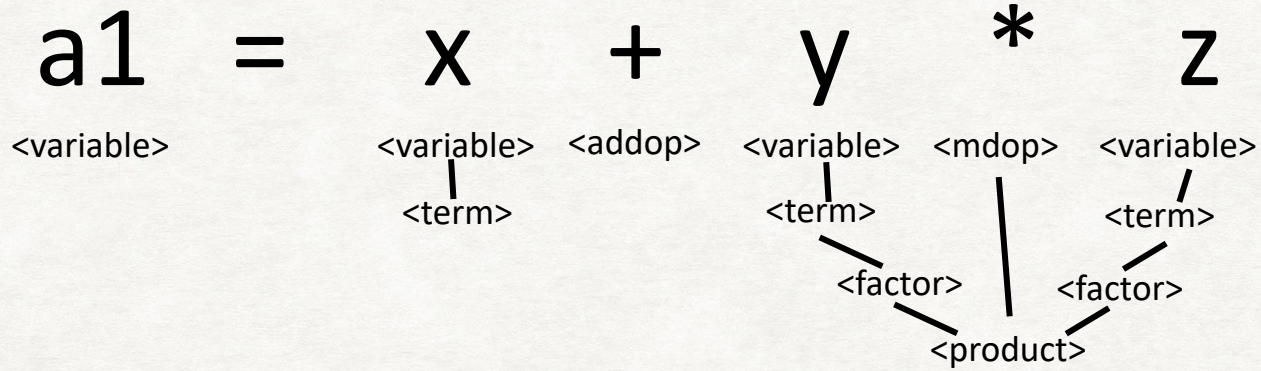
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

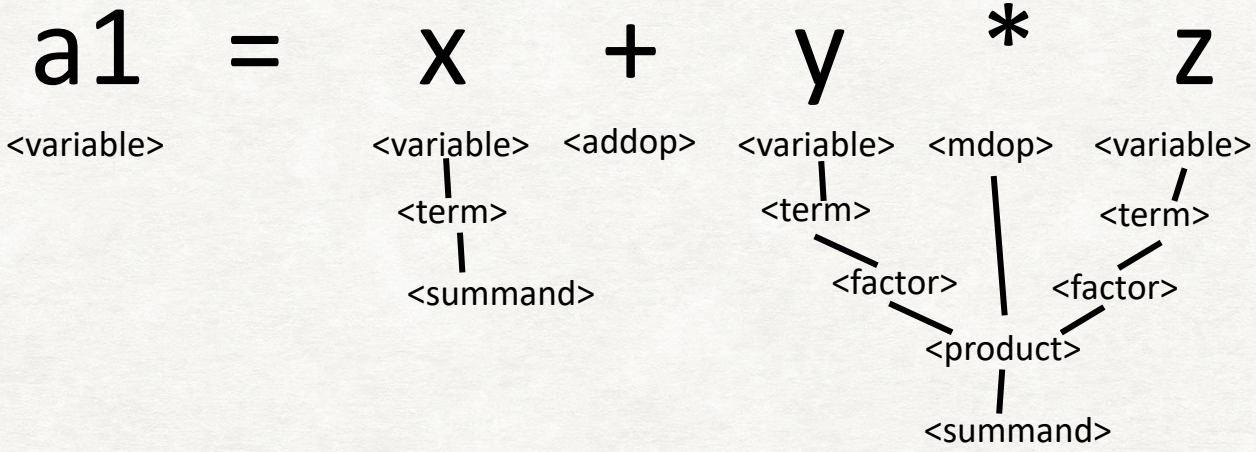
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

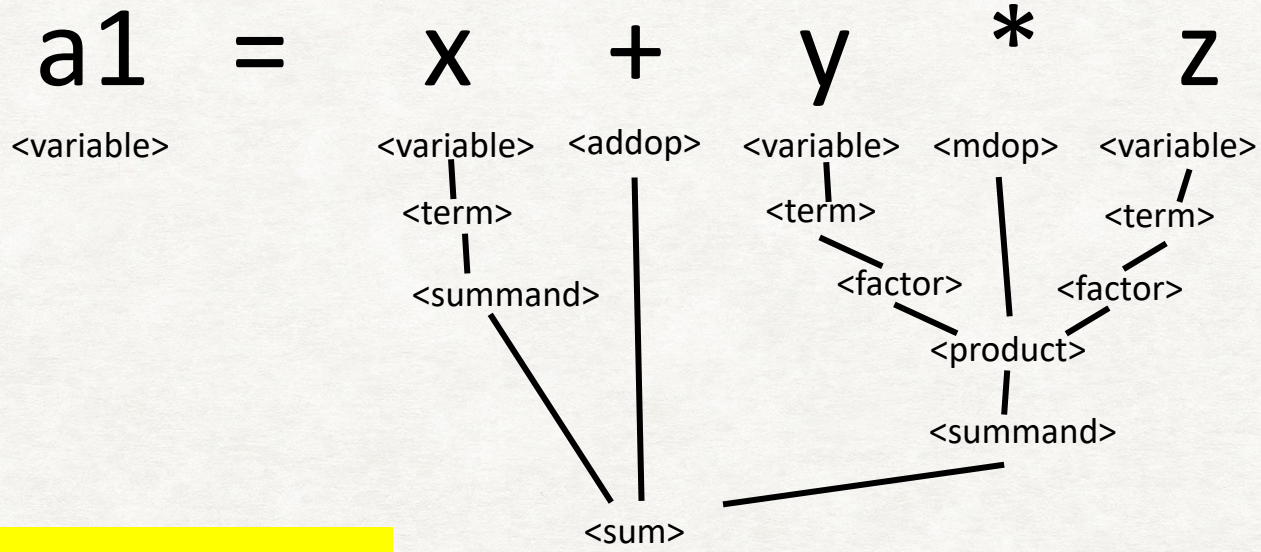
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

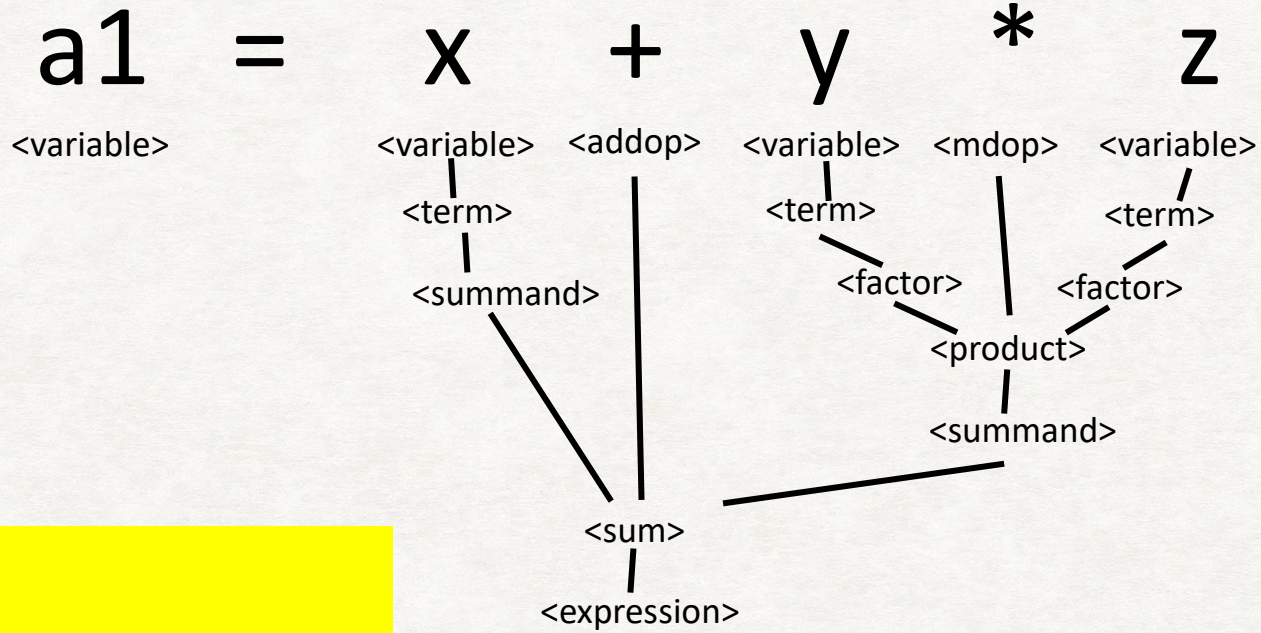
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

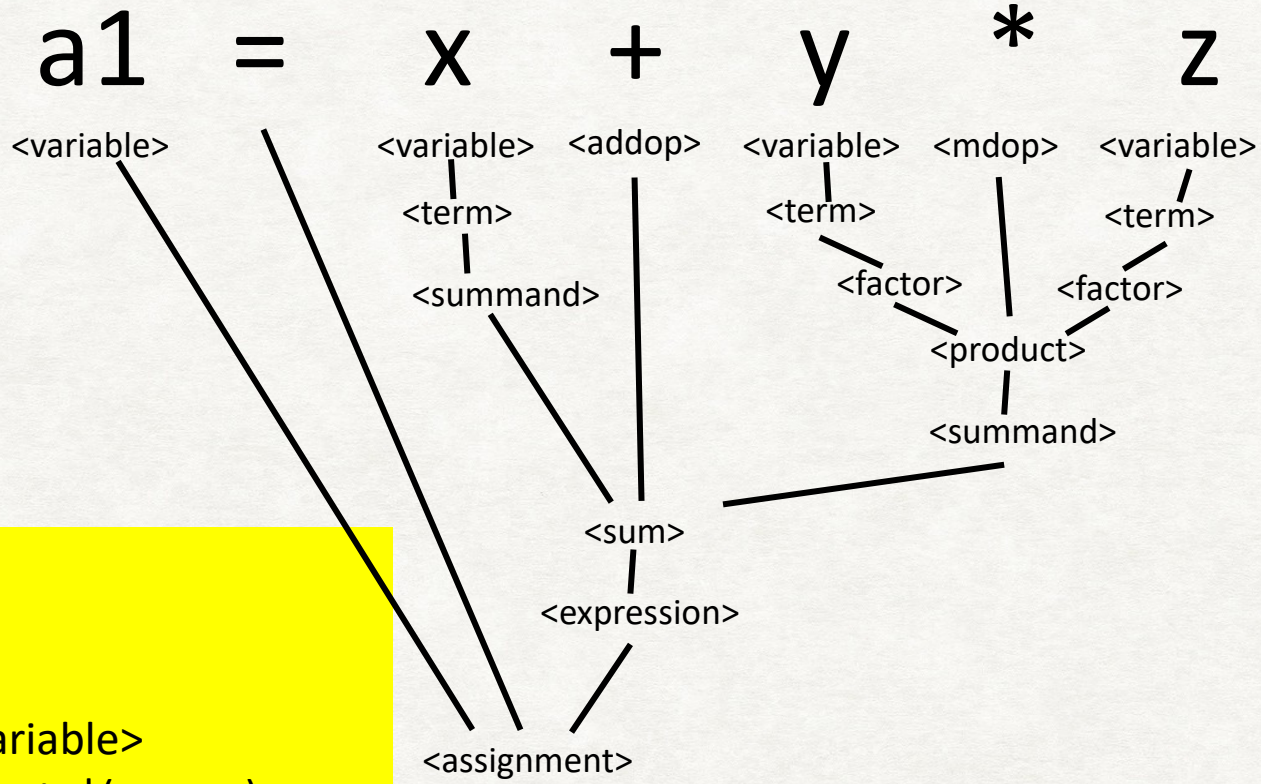
A grammar for multiplication before addition



The problem can be fixed by changing the grammar to make it **unambiguous**

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

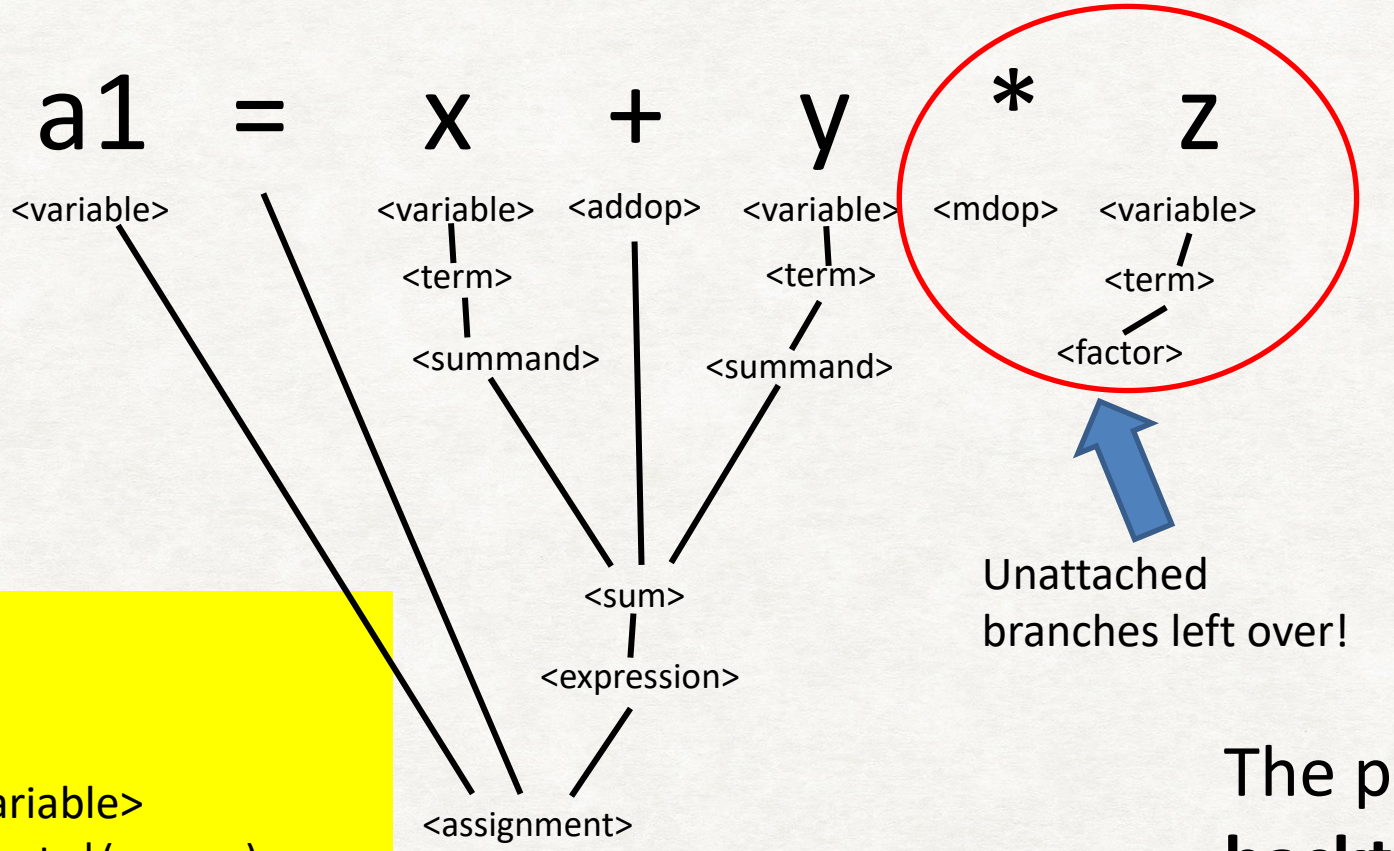
A grammar for multiplication before addition



<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

The problem can be fixed by changing the grammar to make it **unambiguous**

A grammar for multiplication before addition



<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

The parser will **backtrack** (undo the last production) and try to match another grammar rule.

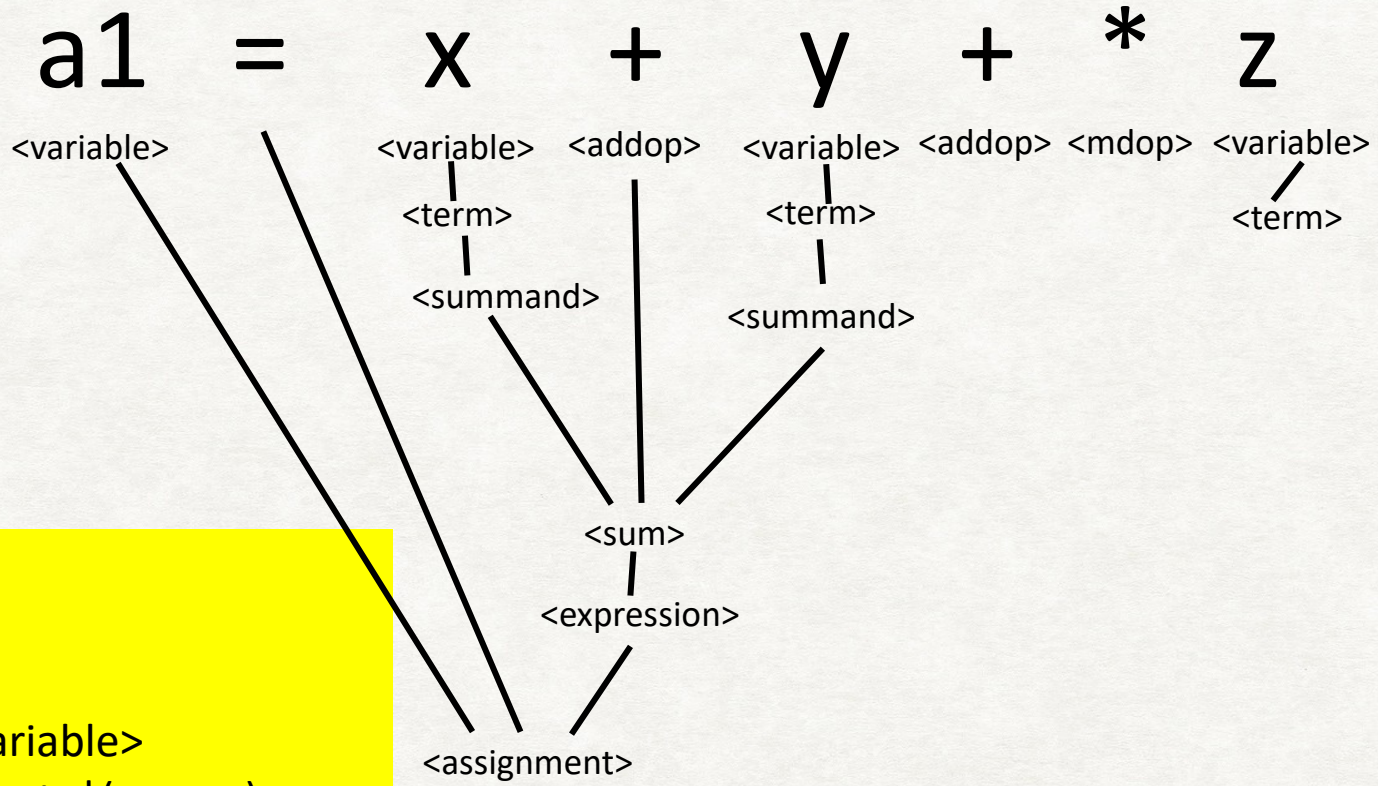
Let’s try parsing another statement.

a1 = x + y + * z

Can’t parse this: No rule in this grammar that allows for any component of an assignment statement to have a + and a * operator next to each other.

<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

Let's try parsing another statement.



<variable> ::= <symbol>
<addop> ::= +|-
<mdop> ::= *|/
<term> ::= <number>|<variable>
<factor> :: <term>|<product>|(<sum>)
<product> ::= <factor><mdop><factor>
<summand> ::= <term>|<product>|<sum>
<sum> :: <summand><addop><summand>
<expression> ::= <term>|<sum>|<product>
<assignment> ::= <variable>=<expression>

Can't parse this: No rule in this grammar that allows for any component of an assignment statement to have a + and a * operator next to each other.

Note: In a different grammar, there could be such a rule.

E.g., the statement above is syntactically correct in C language!

What have we learnt?

If at first we don't succeed in building a parse tree, backtrack and try again with a different combination of rules from the grammar.

- If a parser cannot build a parse tree for a given input and grammar, we have a syntax error.

Grammars *can* be ambiguous – different parse trees are possible depending on the parser implementation!

- In programming languages, grammars *must not* be ambiguous!

Things worth knowing:

Not all grammars use BNF. Some variations in the notation exist. You can find the grammars for a lot of modern high level programming languages in BNF or similar specifications.

For example:

Python: https://docs.python.org/3/reference/grammar.html
Java: https://docs.oracle.com/javase/specs/jls/se11/html/jls-19.html
C#: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/lexical-structure
C++ (gcc): http://www.nongnu.org/hcb/
Not a programming language but still worth having a look at.
PHP: https://github.com/php/php-src/blob/master/Zend/zend_language_parser.y
Ruby: https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/yacc.html

A quick note on grammars

Just because our input is grammatically correct in syntax, it doesn't mean that it makes sense.

> Think English: "Students cannot melt out of musical wiring watermelons" is grammatical but makes absolutely no sense.

In programming languages, we may get code that is grammatically correct, but still wrong:

- Variable names refer to variables that do not exist.
- Function / method names are correct but the number / type of parameters don't match the function / method declaration.
- Variables are of the wrong type. E.g., we try to multiply with a string of letters.

These problems cannot be addressed with a grammar.